Práctica 2: Búsquedas con trayectorias múltiples
Selección de características

Primero el mejor, enfriamiento simulado y búsqueda tabú básica

Alejandro García Montoro
76628233F, agarciamontoro@correo.ugr.es

Grupo de los viernes a las 17.30

Curso 2015 - 2016

Índice

1. Descripción del problema

La selección de características es una técnica muy usada en problemas de aprendizaje automático.

El aprendizaje automático, visto de una forma muy general, tiene como objetivo clasificar un conjunto de objetos —modelador por una serie de atributos— en clases.

Esta clasificación se aprende desde los datos, pero la selección de los atributos que definen la modelización del objeto puede no ser la más apropiada: en ocasiones hay atributos superfluos o demasiado ruidosos que sería conveniente eliminar. Además, cuantos menos atributos definan un objeto, más rápido y preciso será el aprendizaje. Es aquí entonces donde aparece la pregunta que guia todo este trabajo: ¿cómo identificar los atributos que mejor aprendizaje promueven?

La respuesta a esta pregunta pasa por la selección de características, cuyo objetivo es reducir la definición de un objeto a una serie de características que faciliten el aprendizaje.

La idea es entonces la siguiente: dado un conjunto de m objetos definidos por un conjunto C de n características, y considerado un modelo de aprendizaje f que intenta aprender la clasificación de estos objetos, encontrar el subconjunto $C' \subset C$ que maximiza el modelo f.

Así, vemos claramente que el tamaño de caso de nuestro problema es n —el número de características— y que el objetivo está bien definido: eliminar aquellas características que o bien empeoren la bondad de f o bien sean innecesarias.

Con todos estos elementos definidos, podemos pasar a analizar las metaheurísticas consideradas.

2. Metaheurísticas

2.1. Introducción

Los algoritmos considerados para resolver el problema son los siguientes:

- Búsqueda multiarranque básica (BMB).
- *Greedy randomized adaptive search procedure (GRASP).*
- Búsqueda local reiterada (*ILS*).

Además, compararemos estas metaheurísticas con el algoritmo voraz *Sequential forward selection.*

Estas tres metaheurísticas reúnen las condiciones necesarias para resolver el problema: el espacio de soluciones de nuestro problema puede ser analizado mediante las estructuras de generación de vecinos y los criterios de aceptación que utilizan estos algoritmos. Veamos con un poco más de detalle los aspectos comunes a las metaheurísticas implementadas:

Datos de entrada

Todos los algoritmos considerados reciben un conjunto de entrenamiento cuyos objetos tienen la siguiente estructura:

$$(s_1, s_2, \ldots, s_n, c)$$

donde $(s_1, s_2, \ldots, s_n)$ es el conjunto de valores de los atributos que definen el objeto y c la clase a la que pertenece.

Esquema de representación

El espacio de soluciones S de nuestro problema es el conjunto de todos los vectores s de longitud n —el número de características— binarios; es decir:

$$S = \{s = (s_1, s_2, \ldots, s_n)/s_i \in \{0, 1\}\ \forall i = 1, 2, \ldots, n\}$$

La posición i-ésima de un vector $s \in S$ indicará la inclusión o no de la característica i-ésima en el conjunto final C'.

Función objetivo

La finalidad de las metaheurísticas será maximizar la función objetivo siguiente:

$$\begin{aligned} f\colon S &\to [0, 100] \\ s &\mapsto f(s) = \text{Acierto del 3-NN sobre s} \end{aligned}$$

$f(s)$ es, por tanto, la tasa de acierto del clasificador 3-NN producido a partir de la solución s.

El clasificador 3-NN es una particularización del clasificador k-NN, que mide la distancia de la instancia considerada a todos los demás objetos en el conjunto de datos de entrenamiento y le asigna la clasificación mayoritaria de entre los k vecinos más cercanos; esto es:

Pseudocódigo 1 Clasificador k-NN

```
1: function k-NN(instance, trainingData)
2:     distances ← euclideanDistance(instance, trainingData)
3:     neighbours ← getClosestNeighbours(distances)
4:     classification ← mostVotedClassification(neighbours)
5:     return classification
```

Así, dada una solución $s \in S$, la función objetivo es como sigue:

Pseudocódigo 2 Función objetivo

```
1: function f(s, train, target)
2:     samples ← removeZeroColumns(s, train)
3:     sum ← 0
4:     for instance ∈ samples do
5:         class ← k-NN(instance, samples)
6:         sum ← sum + { 1  if class = actualClass(instance, target)
                       { 0  if class ≠ actualClass(instance, target)
7:     return sum / (number of samples in train)
```

donde *removeZeroColumns(s, train)* elimina la columna i-ésima de *train* si y sólo si $s_i = 0$ y *actualClass(instance, target)* devuelve la clase real —no la aprendida— del objeto *instance*.

Entorno de soluciones

Dada una solución $s \in S$, el entorno de soluciones vecinas a s es el conjunto

$$E(s) = \{s' \in S / |s' - s| = (0, \ldots, 0, \underbrace{1}_{i}, 0, \ldots, 0), i \in \{1, 2, \ldots, n\}\}$$

es decir, $E(s)$ son las soluciones que difieren de s en una única posición. Es evidente entonces que el conjunto $E(S)$ tiene siempre exactamente cardinal igual a n.

El operador de generación de vecino de la solución s es entonces como sigue:

Pseudocódigo 3 Operador de generación de vecino

1: **function** FLIP(solution, feature)
2: $s' \leftarrow solution$
3: $s'[feature] \leftarrow (s'[feature] + 1) \bmod 2$
4: **return** s'

Criterios de parada

Aunque los criterios de parada dependerán de la metaheurística considerada —en general se parará cuando no se encuentre mejora en el entorno—, en todos los algoritmos pararemos necesariamente tras llegar a las 15000 evaluaciones con el clasificador 3-NN sobre las soluciones generadas.

Generación de soluciones aleatorias

En los algoritmos de búsqueda multiarranque básica y búsqueda local reiterada se genera una serie de soluciones aleatorias sobre las que se aplica búsqueda local de una u otra forma. La generación de estas soluciones aleatorias sigue el siguiente esquema:

Pseudocódigo 4 Generación de soluciones aleatorias

1: **function** RANDOMSOLUTION(size)
2: **for** $i \in 1, 2, \ldots, size$ **do**
3: random $\leftarrow$ uniformRandomNumber([0,1])
4: $s_i \leftarrow \begin{cases} 0 & \textbf{if } random \leq 0{,}5 \\ 1 & \textbf{if } random > 0{,}5 \end{cases}$
5: solution $\leftarrow (s_1, s_2, \ldots, s_{size})$
6: **return** solution

Búsqueda local

El algoritmo de búsqueda local considerado es el implementado para la primera práctica: la búsqueda local primero el mejor.

El método de exploración del entorno es el siguiente: dada una solución s, escogemos una característica al azar, aplicamos el operador $flip$ para obtener una solución vecina y medimos su bondad con $f(s)$; si es mejor que s, nos quedamos con ella como mejor solución y volvemos a empezar; si no, tomamos otra característica al azar —sin repetir— y seguimos el proceso.

Pararemos el algoritmo si: o bien al haber explorado el entorno completo de la solución actual ninguna de las soluciones vecinas es mejor, o bien si se han alcanzado 15000 iteraciones. Estaremos entonces ante un máximo —probablemente local— y el algoritmo no podrá seguir mejorando la solución.

El pseudocódigo de todo el procedimiento es el siguiente, donde hemos puesto un tercer argumento optativo, *initSol*, que por defecto es vacío, y en cuyo caso se genera una solución aleatoria como solución inicial. Si no es vacío, tomamos como solución inicial la *initSol*, de manera que la búsqueda se centrará en el entorno de esta solución.

Pseudocódigo 5 Búsqueda local primero el mejor

```
function bestFirst(train, target, initSol = ∅)
    if initSol = ∅ then
        s ← genInitSolution()
    else
        s ← initSol
    bestScore ← f(s, train, target)
    improvementFound ← True
    while improvementFound and iterations <15000 do
        improvementFound ← False
        for feature ← genRandomFeature(s) do    ▷ Without replacement
            s'← genNeighbour(s, feature)
            score ← f(s', train, target)
            if score > bestScore then
                s, bestScore ← s', score
                improvementFound ← True
                break for
    return s, bestScore
```

2.2. Búsqueda multiarranque básica

La búsqueda multiarranque básica es el primer algoritmo considerado, cuyo comportamiento es muy sencillo: se trata de generar un número N de soluciones aleatorias y, para cada una de ellas, explotar su entorno de soluciones con el algoritmo de búsqueda local primero el mejor.

La idea que intenta perseguir este algoritmo es clara: como la búsqueda local ya aporta la suficiente intensificación en zonas locales del espacio de búsqueda, se intenta aumentar la diversidad para explorar zonas diferentes. Esto último se consigue con la generación aleatoria de las N soluciones, que previsiblemente cubren una zona mayor del espacio.

Durante todo el proceso mantendremos la mejor solución encontrada hasta el momento, de manera que al terminar el algoritmo —tras buscar localmente desde 25 soluciones aleatorias y con un máximo de 15000 iteraciones en cada búsqueda local— se devuelve la mejor encontrada.

Como el procedimiento *bestFirst* genera una solución aleatoria inicial si no se le pasa un tercer argumento, todo el cómputo del algoritmo BMB se encuentra recogido en esa llamada. Sólo tenemos que preocuparnos de repetir

el procedimiento N veces, donde N es, en este caso, igual a 25.

El procedimiento se puede ver en el Pseudocódigo 6.

Pseudocódigo 6 Búsqueda multiarranque básica

```
1: function BMB(train, target)
2:     bestSolution, bestScore ← ∅, -1
3:     for i ∈ {1, 2, ..., 25} do
4:         currentSolution, currentScore ← bestFirst(train, target)
5:         if currentScore >bestScore then
6:             bestSolution, bestScore ← currentSolution, currentScore
7:     return bestSolution, bestScore
=0
```

2.3. *Greedy randomized adaptive search procedure*

La estructura del algoritmo GRASP es similar a la de la búsqueda multiarranque básica: tras generar una solución inicial, se ejecuta búsqueda local sobre ella para mejorarla. La diferencia con el algoritmo anterior se encuentra, precisamente, en cómo se genera la solución inicial: si bien en BMB se hacía de forma aleatoria, aquí se añade un paso intermedio: tras generar una solución aleatoria, se ejecuta una variante del algoritmo *SFS* que en cada iteración elige, de entre las características que más ganancia aportan, una de forma aleatoria.

Antes de entrar en detalles, es conveniente ver el procedimiento general en el Pseudocódigo 7.

Pseudocódigo 7 GRASP

```
1: function BMB(train, target)
2:     bestSolution, bestScore ← ∅, -1
3:     for i ∈ {1, 2, ..., 25} do
4:         currentSolution, currentScore ← randomSFS(train, target)
5:         currentSolution, currentScore ← bestFirst(train, target, curren-
   tSolution)
6:         if currentScore >bestScore then
7:             bestSolution, bestScore ← currentSolution, currentScore
8:     return bestSolution, bestScore
```

Como vemos, sólo se ha añadido la generación de la solución inicial antes de la búsqueda local. En la llamada a *randomSFS* se encuentra el núcleo de este algoritmo, así que echémosle un vistazo más de cerca.

Algoritmo voraz probabilístico

La generación de la solución inicial en el algoritmo *GRASP* se hace con un algoritmo voraz probabilístico, cuya idea central es la siguiente: en cada iteración se evalúan todas las características aún no seleccionadas, almacenando para cada una de ellas la ganancia —que puede ser negativa— que produce con respecto a la solución actual; tras esta evaluación, se genera una lista restringida de candidatos definida a partir de la peor y mejor ganancia registradas; de entre esta lista, se toma una característica al azar y se añade a la solución, terminando así la iteración.

La generación de la lista restringida de candidatos, LRC, se hace en base al siguiente umbral:

$$\mu = \max_i\{g_i\} - \alpha(\max_i\{g_i\} - \min_i\{g_i\})$$

donde g_i es la ganancia que produce añadir la característica i-ésima a la solución actual y, en este caso, se ha tomado $\alpha = 0{,}3$.

Podemos ya ver el procedimiento del algoritmo voraz probabilístico en el Pseudocódigo 8.

Pseudocódigo 8 Algoritmo voraz probabilístico

```
1:  function RANDOMSFS(train, target)
2:      s ← genZeroSolution()
3:      bestScore ← 0
4:      while there was improvement with some feature do
5:          g ← (0, 0, ..., 0)          ▷ Size = number of not selected features
6:          for every feature f in not selected features do
7:              s ← addFeature(s,f)
8:              currentScore ← f(s, train, target)
9:              gain ← currentScore - bestScore
10:             g_f ← gain
11:             s ← removeFeature(s,f)
12:         μ ← máx_i{g_i} − α(máx_i{g_i} − mín_i{g_i})
13:         LRC ← {f ∈ not selected features / g_f > μ}
14:         f ← random choice from LRC
15:         if g_f > 0 then
16:             s ← addFeature(s,f)
17:             bestScore ← bestScore + bestGain
18:     return s, bestScore
```

2.4. Búsqueda local reiterada

El algoritmo de búsqueda local reiterada, o ILS por sus siglas en inglés, busca también introducir algo de diversidad en las soluciones exploradas.

Para ello, parte de una solución inicial aleatoria que mejora con búsqueda local. A partir de ahí se sigue un procedimiento iterativo repetido N veces —en nuestro caso, tomaremos $N = 24$, ya que queremos llamar 25 veces a la búsqueda local—:

- Se muta la solución previa.
- Se realiza búsqueda local con esa mutación como solución inicial.
- Se actualiza la mejor solución.

Podemos ver este procedimiento con más detalle en el Pseudocódigo 9.

Pseudocódigo 9 Búsqueda local reiterada

```
1: function ILS(train, target)
2:     bestSolution, bestScore ← bestFirst(trian, target)
3:     prevSolution, prevScore ← bestSolution, bestScore
4:     for i ∈ {1, 2, ..., 24} do
5:         mutation ← mutateSolution(prevSolution)
6:         currentSolution, currentScore ← bestFirst(train, target, currentSolution)
7:         if currentScore >prevScore then
8:             prevSolution, prevScore ← currentSolution, currentScore
9:         if prevScore >bestScore then
10:            bestSolution, bestScore ← prevSolution, prevScore
11:    return bestSolution, bestScore
```

Mutación

La mutación de las soluciones llevada a cabo en el algoritmo ILS es simple: basta tomar un 10 % de características de forma aleatoria y cambiar su estado: si están a 1 ponerlas a 0 y viceversa; esto es, aplicar el operador *flip* para cada una de ellas.

Este sencillo procedimiento puede verse en el Pseudocódigo 10.

Pseudocódigo 10 Mutación para la ILS

```
1: function MUTATESOLUTION(solution, p = 0.1)
2:     n ← size of solution
3:     indices ← Take ⌈pn⌉ random indices in {1, ..., n}
4:     for i ∈ indices do
5:         flip(solution, i)
6:     return solution
```

2.5. Algoritmo de comparación

Para la comparación de los algoritmos implementados consideraremos el algoritmo voraz *Sequential forward selection*, que se puede ver en el Pseudocódigo 11.

Pseudocódigo 11 Algoritmo de comparación

```
1:  function sequentialForwardSelection(train, target)
2:      s ← genZeroSolution()
3:      bestScore ← 0
4:      while there was improvement with some feature do
5:          for every feature f in not selected features do
6:              s ← addFeature(s,f)
7:              currentScore ← score(s, train, target)
8:              if currentScore > bestScore then
9:                  bestScore ← currentScore
10:                 bestFeature ← f
11:             s ← removeFeature(s,f)
12:         if there was a best feature f then
13:             s ← addFeature(s,f)
14:     return s, bestScore
```

La idea es la siguiente: en cada iteración escogemos la característica, de entre las aún no seleccionadas, que mejor valor de la función objetivo produce, si y sólo si este valor es mejor que el actual.

3. Desarrollo de la práctica

La práctica se ha desarrollado por completo en Python, definiendo cada algoritmo en una función diferente con cabeceras iguales —mismo número y tipo de parámetros— para poder automatizar el proceso de recogida de datos.

3.1. *Framework* de aprendizaje automático

Se ha usado, además, el módulo *Scikit-learn*, del que se ha usado la siguiente funcionalidad:

- Particionamiento de los datos. *Scikit-learn* aporta una función para hacer un particionado aleatorio de los datos en una parte de aprendizaje y otra de test. Esto se ha usado para implementar la técnica 5×2 *cross-validation*.

3.2. Paralelización en GPU de la función objetivo

Aunque en la práctica anterior se usó también *Scikit-learn* para medir la función objetivo, la lentitud de este proceso me llevó a buscar otras alternativas: después de intentar usar el mismo módulo con la opción de paralelización CPU y conseguir prácticamente los mismos resultados —para notar mejoría, dicen los desarrolladores, haría falta trabajar con bases de datos con varios miles de muestras—, decidí buscar una solución propia.

Como gracias a mi Trabajo fin de grado he aprendido a hacer computación general paralelizada en GPU, decidí usar la librería CUDA —y en concreto su interfaz para Python, PyCUDA— para implementar la función objetivo de una forma eficiente. La mejoría en tiempo conseguida es muy notable —es del orden de 20 a 100 veces más rápido[1]— y, tras muchas pruebas para comprobar que el cálculo de la función era correcto, sustituí el k-NN de *Scikit-learn* con el implementado en CUDA.

Todo este trabajo, necesario para el correcto funcionamiento de la práctica, se encuentra en los ficheros bajo el directorio *src/knnGPU*, que contienen la implementación en C del k-NN y la interfaz para poder usar el código desde Python.

Además, como vi que este código podía beneficiar a mis compañeros, decidí publicarlo de forma abierta en un repositorio de Github[2], bien documentado y con una guía de uso.

Gracias a esto, algunos amigos me ayudaron a mejorar el código: yo había implementado sólo la función objetivo sobre los datos de training, y Jacinto

[1] Los tiempos son muy dependientes del número de muestras de la base de datos y del número de características. Para tener una idea de la mejora, se pueden comparar los tiempos de las tablas 3-NN y SFS de esta y la anterior práctica.

[2] `https://github.com/agarciamontoro/metaheuristics`

Carrasco Castillo la modificó para poder hacer la medición también con los datos de test. Además, Luís Suárez Lloréns me ayudó a probar cambios que creíamos que iban a mejorar aún más la eficiencia —aunque tras mucho trabajo vimos que la implementación inicial era la más rápida—. Por último, Antonio Álvarez Caballero, Anabel Gómez Ríos y Gustavo Rivas Gervilla me ayudaron a testear el código, probándolo con sus algoritmos y los datos que tenían de anteriores prácticas.

3.3. Manual de usuario

Para la ejecución de la práctica es necesario tener instalado Python 3, el módulo *Scikit-learn*, *PyCUDA* y *jinja2* —estos dos últimos módulos son necesarios para la implementación del código paralelizado—, así como disponer de una tarjeta gráfica compatible con CUDA.

Todo se encuentra automatizado en el fichero *src/02_multiPath.py*, así que sólo es necesario ejecutar la siguiente orden desde el directorio raíz de la práctica: *python src/02_multiPath.py*

Así se ejecutarán todos los algoritmos con todas las bases de datos usando la ténica del 5×2 *cross-validation*. Las tablas generadas se guardarán en el directorio *results/02*.

La semilla utilizada se inicializa al principio de la ejecución del programa con las líneas *np.random.seed(19921201)* y *random.seed(19921201)*.

4. Análisis de resultados

En esta sección vamos a presentar los datos recogidos de la ejecución de todos los algoritmos con las tres bases de datos consideradas: *WDBC*, *Movement Libras* y *Arrhytmia*. Las bases de datos se han considerado completas en todos los casos, tal y como se nos entregaron —arreglando alguna columna defectuosa y homogeneizando el nombre de la columna de clasificación para poder automatizar el proceso—.

Para el análisis de cada algoritmo con cada base de datos se han generado cinco particiones aleatorias de los datos y se ha ejecutado el algoritmo considerando cada partición como datos de entrenamiento y test, con la técnica 5×2 *cross-validation*.

En cada una de estas ejecuciones se han medido los siguientes datos:

- Tasa de clasificación en la partición de entrenamiento —en %—.
- Tasa de clasificación en la partición de test —en %—.
- Tasa de reducción de las características —en %—.
- Tiempo de ejecución —en segundos—.

Veamos ya los datos y analicemos los resultados obtenidos:

4.1. Clasificador k-NN

	WDBC				Movement Libras				Arrythmia			
Particiones	%Clas. in	%Clas. out	%Red.	T	%Clas. in	%Clas. out	%Red.	T	%Clas. in	%Clas. out	%Red.	T
Partición 1-1	97,1831	96,4912	0	$4{,}7 \cdot 10^{-3}$	68,8889	71,1111	0	$4{,}7 \cdot 10^{-3}$	64,0625	60,3093	0	0,0201
Partición 1-2	96,4912	96,1268	0	$3{,}2 \cdot 10^{-3}$	72,2222	75,5556	0	$4{,}7 \cdot 10^{-3}$	60,8247	63,5417	0	0,0184
Partición 2-1	96,4789	96,8421	0	$3{,}3 \cdot 10^{-3}$	72,2222	73,3333	0	$4{,}7 \cdot 10^{-3}$	62,5	61,8557	0	0,0195
Partición 2-2	96,8421	97,1831	0	$3{,}4 \cdot 10^{-3}$	73,3333	75	0	$4{,}7 \cdot 10^{-3}$	60,8247	65,625	0	0,0182
Partición 3-1	96,831	96,4912	0	$3{,}3 \cdot 10^{-3}$	70	77,7778	0	$4{,}7 \cdot 10^{-3}$	61,9792	64,433	0	0,0196
Partición 3-2	97,193	96,4789	0	$3{,}7 \cdot 10^{-3}$	66,6667	76,1111	0	$4{,}7 \cdot 10^{-3}$	63,9175	63,0208	0	0,0181
Partición 4-1	95,0704	96,1404	0	$3{,}5 \cdot 10^{-3}$	68,3333	63,3333	0	$4{,}7 \cdot 10^{-3}$	64,0625	62,3711	0	0,0195
Partición 4-2	95,7895	96,4789	0	$3{,}5 \cdot 10^{-3}$	71,1111	75	0	$4{,}6 \cdot 10^{-3}$	62,3711	61,9792	0	0,0184
Partición 5-1	95,7746	95,0877	0	$3{,}3 \cdot 10^{-3}$	65,5556	77,7778	0	$4{,}7 \cdot 10^{-3}$	65,1042	61,8557	0	0,0196
Partición 5-2	96,8421	96,4789	0	$3{,}3 \cdot 10^{-3}$	73,3333	73,3333	0	$4{,}6 \cdot 10^{-3}$	60,8247	67,1875	0	0,0181
Medias	96,4496	96,3799	0	$3{,}5 \cdot 10^{-3}$	70,1667	73,8333	0	$4{,}7 \cdot 10^{-3}$	62,6471	63,2179	0	0,019

Cuadro 1: Datos del clasificador k-NN

En la tabla 1 se pueden ver los datos obtenidos del clasificador k-NN. La selección de características en este algoritmo es nula, ya que es la propia función objetivo considerando la totalidad de las características. Aún así, se ha añadido aquí para conocer la tasa de clasificación en los conjuntos de entrenamiento y de test considerando como solución la trivial: esto es, todas las características.

Como vemos, aunque en la primera base de datos las tasas de clasificación son buenas, en las otras dos son muy mejorables, lo que nos da una idea de la necesidad de la reducción de características.

4.2. Algoritmo de comparación

	WDBC				Movement Libras				Arrythmia			
Particiones	%Clas. in	%Clas. out	%Red.	T	%Clas. in	%Clas. out	%Red.	T	%Clas. in	%Clas. out	%Red.	T
Partición 1-1	97,8873	95,4386	83,3333	0,2453	78,8889	65,5556	90	0,9302	84,8958	68,0412	96,4029	3,495
Partición 1-2	97,5439	93,3099	80	0,2493	76,6667	70,5556	91,1111	0,8259	81,9588	68,75	96,0432	3,8653
Partición 2-1	97,5352	95,4386	86,6667	0,1729	75,5556	70,5556	90	0,9302	84,375	73,1959	96,7626	3,0406
Partición 2-2	95,0877	94,3662	86,6667	0,1784	80,5556	71,6667	90	0,9252	75,2577	66,6667	98,9209	1,0483
Partición 3-1	95,4225	92,2807	86,6667	0,1742	71,6667	74,4444	91,1111	0,8206	73,9583	70,6186	98,5612	1,3368
Partición 3-2	97,5439	94,7183	83,3333	0,2216	75	72,2222	91,1111	0,8245	82,9897	73,9583	97,1223	2,6323
Partición 4-1	97,1831	92,2807	93,3333	0,0992	75,5556	68,3333	90	0,9266	78,6458	67,5258	98,2014	1,6346
Partición 4-2	97,193	94,3662	76,6667	0,2978	77,7778	72,7778	90	0,9279	72,6804	72,3958	98,5612	1,3186
Partición 5-1	97,8873	94,7368	86,6667	0,1752	69,4444	67,2222	92,2222	0,7209	81,25	76,2887	97,1223	2,6349
Partición 5-2	97,193	95,0704	83,3333	0,2161	81,1111	72,7778	87,7778	1,1426	72,1649	71,3542	99,2806	0,7371
Medias	97,0477	94,2006	84,6667	0,203	76,2222	70,6111	90,3333	0,8975	78,8177	70,8795	97,6978	2,1744

Cuadro 2: Datos del algoritmo *Sequential forward selection*

En la tabla 2 vemos los resultados del algoritmo de comparación, el *Sequential forward selection*. Este algoritmo voraz tiene una alta tasa de reducción de características, pero la tasa de clasificación no mejora la del clasificador con la solución trivial.

Esto se debe a que consideramos cada característica de una forma secuencial, y una vez seleccionamos una, es imposible descartarla. Aún así, este algoritmo podría ser interesante si lo que buscamos es una reducción drástica del número de características —como vemos, sobre el 80 %— sin perder mucha información —las tasas de clasificación son más o menos iguales a las del clasificador con la solución trivial—.

4.3. Algoritmo genético generacional

	WDBC				Movement Libras				Arrythmia			
Particiones	%Clas. in	%Clas. out	%Red.	T	%Clas. in	%Clas. out	%Red.	T	%Clas. in	%Clas. out	%Red.	T
Partición 1-1	98,9437	97,5439	40	37,4253	76,6667	69,4444	57,7778	39,4115	73,9583	62,3711	52,518	106,3711
Partición 1-2	97,8947	96,4789	56,6667	33,2978	78,3333	76,6667	45,5556	45,6983	71,6495	65,1042	48,5612	107,5029
Partición 2-1	98,5915	95,4386	56,6667	33,2878	75	73,3333	55,5556	40,6443	75,5208	64,9485	50,7194	110,3791
Partición 2-2	96,1404	96,4789	40	37,9449	78,8889	75,5556	54,4444	41,2447	71,6495	66,1458	53,5971	101,6543
Partición 3-1	98,2394	94,0351	50	34,582	74,4444	76,6667	54,4444	41,3164	73,4375	63,4021	48,5612	114,3843
Partición 3-2	97,8947	96,831	50	36,6143	77,2222	78,8889	44,4444	45,8163	69,5876	63,0208	53,5971	98,1602
Partición 4-1	97,8873	94,0351	56,6667	33,4136	76,6667	68,8889	55,5556	40,8464	69,7917	63,4021	50	111,3771
Partición 4-2	97,8947	96,4789	46,6667	36,9319	81,1111	74,4444	42,2222	48,3968	71,6495	60,4167	51,4388	99,2755
Partición 5-1	97,5352	94,7368	66,6667	31,1176	75	75	58,8889	39,2112	72,9167	63,9175	49,6403	111,8573
Partición 5-2	97,5439	96,1268	50	35,2995	83,3333	73,3333	57,7778	39,5866	72,6804	69,7917	53,2374	100,8417
Medias	97,8566	95,8184	51,3333	34,9915	77,6667	74,2222	52,6667	42,2172	72,2841	64,252	51,1871	106,1804

Cuadro 3: Datos del algoritmo genético generacional

	WDBC				Movement Libras				Arrythmia			
Particiones	%Clas. in	%Clas. out	%Red.	T	%Clas. in	%Clas. out	%Red.	T	%Clas. in	%Clas. out	%Red.	T
Partición 1-1	97,8873	95,4386	33,3333	38,8089	76,6667	72,7778	50	42,9188	72,3958	66,4948	49,6403	121,0009
Partición 1-2	96,8421	95,0704	40	36,6349	77,7778	72,7778	51,1111	43,2273	72,6804	69,2708	52,518	110,877
Partición 2-1	97,1831	95,4386	53,3333	33,4498	80,5556	69,4444	50	43,0703	73,4375	63,4021	53,9568	112,7747
Partición 2-2	97,5439	96,4789	43,3333	36,5362	77,2222	70	46,6667	45,2997	75,7732	65,625	52,1583	114,2411
Partición 3-1	97,1831	94,7368	50	34,4817	73,3333	73,3333	58,8889	39,1173	72,9167	62,8866	50	125,4697
Partición 3-2	96,4912	94,3662	50	33,5299	81,6667	74,4444	60	37,7801	74,2268	66,6667	51,7986	116,2933
Partición 4-1	97,8873	95,7895	60	32,7299	75,5556	68,8889	44,4444	46,6692	72,9167	58,7629	55,3957	113,7118
Partición 4-2	96,1404	97,5352	43,3333	35,8729	77,7778	73,3333	55,5556	41,2552	72,1649	65,625	54,6763	104,6554
Partición 5-1	96,4789	98,2456	53,3333	33,4025	76,6667	76,6667	51,1111	42,6229	76,0417	68,5567	53,2374	116,0981
Partición 5-2	97,193	91,1972	66,6667	30,2318	81,1111	78,8889	50	43,7381	67,0103	66,1458	54,6763	114,6064
Medias	97,083	95,4297	49,3333	34,5679	77,8333	73,0556	51,7778	42,5699	72,9564	65,3436	52,8058	114,9728

Cuadro 4: Datos del algoritmo genético generacional con cruce HUX

En la tabla **??** se encuentran los datos referentes a la ejecución del algoritmo GRASP sobre todas las bases de datos.

Vemos cómo este algoritmo consigue unas tasas de reducción bastante altas y, a la vez, unas clasificaciones en la función objetivo mejores que los algoritmos anteriores.

Esto se debe a que este algoritmo reúne las bondades del algoritmo voraz —alta reducción de características— y de la búsqueda local —mejores clasificaciones en la función objetivo—, consiguiendo así ya unos resultados más aceptables.

Además, este algoritmo es más rápido que BMB, así que su comportamiento —tanto en resultados como en eficiencia— es el más deseable de entre los vistos hasta ahora.

4.4. Algoritmo genético estacionario

	WDBC				Movement Libras				Arrythmia			
Particiones	%Clas. in	%Clas. out	%Red.	T	%Clas. in	%Clas. out	%Red.	T	%Clas. in	%Clas. out	%Red.	T
Partición 1-1	97,5352	96,1404	30	40,1997	77,2222	71,1111	53,3333	41,2107	77,0833	64,433	48,9209	114,9172
Partición 1-2	97,193	95,7746	30	40,2496	80	77,7778	46,6667	44,2017	76,8041	69,2708	53,2374	103,6067
Partición 2-1	97,5352	95,4386	43,3333	37,3617	76,1111	72,7778	54,4444	41,45	77,0833	63,9175	45,6835	132,6037
Partición 2-2	96,8421	96,4789	46,6667	35,7629	81,6667	74,4444	66,6667	34,1837	75,2577	64,0625	55,036	96,4863
Partición 3-1	98,2394	96,8421	60	33,0309	78,3333	77,2222	57,7778	38,8077	77,0833	67,5258	50,3597	109,141
Partición 3-2	97,8947	97,1831	56,6667	33,9882	79,4444	75	56,6667	39,3317	77,3196	65,625	51,0791	104,7168
Partición 4-1	97,8873	94,0351	56,6667	33,346	78,3333	67,2222	52,2222	42,4324	75,5208	68,0412	52,8777	107,022
Partición 4-2	97,5439	96,1268	50	37,314	80	75	57,7778	38,8561	75,2577	61,9792	53,9568	107,5883
Partición 5-1	97,1831	95,0877	40	38,4433	76,6667	74,4444	54,4444	40,6448	76,0417	63,4021	48,9209	112,286
Partición 5-2	97,8947	97,1831	30	40,6782	83,8889	73,8889	48,8889	43,2292	75,7732	63,5417	54,3165	101,8523
Medias	97,5749	96,029	44,3333	37,0375	79,1667	73,8889	54,8889	40,4348	76,3225	65,1799	51,4388	109,022

Cuadro 5: Datos del algoritmo genético estacionario

	WDBC				Movement Libras				Arrythmia			
Particiones	%Clas. in	%Clas. out	%Red.	T	%Clas. in	%Clas. out	%Red.	T	%Clas. in	%Clas. out	%Red.	T
Partición 1-1	97,8873	96,8421	40	38,6422	77,2222	73,3333	46,6667	44,3889	79,1667	65,4639	54,6763	108,0894
Partición 1-2	97,193	94,7183	46,6667	34,7281	81,1111	71,6667	42,2222	46,9419	73,7113	67,7083	51,4388	119,086
Partición 2-1	96,831	95,0877	53,3333	35,582	83,3333	71,1111	56,6667	39,6622	77,6042	68,0412	51,7986	117,0812
Partición 2-2	98,2456	95,0704	46,6667	35,5618	80	73,8889	46,6667	43,9596	76,2887	61,9792	53,2374	109,3596
Partición 3-1	97,1831	95,4386	46,6667	35,6276	75,5556	74,4444	52,2222	42,1211	73,9583	61,8557	53,9568	109,3557
Partición 3-2	97,193	95,7746	56,6667	32,6036	83,3333	77,7778	51,1111	43,1308	75,7732	61,4583	53,9568	116,0101
Partición 4-1	97,8873	95,4386	40	37,5466	78,3333	66,6667	58,8889	38,3152	78,125	61,3402	51,0791	114,2392
Partición 4-2	96,1404	96,4789	46,6667	37,1786	78,8889	72,2222	56,6667	39,8531	76,2887	69,7917	57,9137	98,0019
Partición 5-1	96,4789	96,4912	40	36,0821	78,8889	77,7778	44,4444	45,4641	75,5208	63,4021	48,2014	121,6686
Partición 5-2	97,8947	96,4789	36,6667	37,8371	82,2222	77,2222	51,1111	43,5446	74,7423	69,2708	55,3957	106,6866
Medias	97,2934	95,7819	45,3333	36,139	79,8889	73,6111	50,6667	42,7381	76,1179	65,0311	53,1655	111,9578

Cuadro 6: Datos del algoritmo genético estacionario con cruce HUX

En la tabla **??** vemos los datos de la última metaheurística considerada: la búsqueda local reiterada.

Los resultados de este algoritmo deberían ser parecidos a los de la búsqueda multiarranque básica, ya que la idea detrás de su funcionamiento es parecida. A la vista de los resultados, es evidente que así es.

En este caso, sin embargo, se intenta añadir diversidad con la mutación de las soluciones. Esta diversidad es bastante positiva en lo referente a los tiempos: ILS tarda menos, en promedio, que BMB en encontrar las soluciones. Esto puede deberse a que las soluciones iniciales están más cerca

de los óptimos locales, y la búsqueda local necesita de menos iteraciones para alcanzarla. ¿Por qué pasa esto? Al estar en zonas buenas del espacio de búsqueda, es probable que la mutación de la mejor solución encontrada en la iteración anterior se mueva a zonas mejores, haciendo que la búsqueda local converja más rápido.

Esta característica es muy positiva, ya que conseguimos resultados muy similares al BMB con una mejora en la eficiencia.

4.5. Datos generales

	WDBC				Movement Libras				Arrythmia			
Particiones	%Clas. in	%Clas. out	%Red.	T	%Clas. in	%Clas. out	%Red.	T	%Clas. in	%Clas. out	%Red.	T
3-NN	96,4496	96,3799	0	$3{,}5 \cdot 10^{-3}$	70,1667	73,8333	0	$4{,}7 \cdot 10^{-3}$	62,6471	63,2179	0	0,019
SFS	97,0477	94,2006	84,6667	0,203	76,2222	70,6111	90,3333	0,8975	78,8177	70,8795	97,6978	2,1744
AGG	97,8566	95,8184	51,3333	34,9915	77,6667	74,2222	52,6667	42,2172	72,2841	64,252	51,1871	106,1804
AGG-HUX	97,083	95,4297	49,3333	34,5679	77,8333	73,0556	51,7778	42,5699	72,9564	65,3436	52,8058	114,9728
AGE	97,5749	96,029	44,3333	37,0375	79,1667	73,8889	54,8889	40,4348	76,3225	65,1799	51,4388	109,022
AGE-HUX	97,2934	95,7819	45,3333	36,139	79,8889	73,6111	50,6667	42,7381	76,1179	65,0311	53,1655	111,9578

Cuadro 7: Datos generales

En la tabla 7 vemos un resumen de todos los datos obtenidos tras las ejecuciones de las metaheurísticas con las bases de datos.

Vemos ahora claro que el mejor algoritmo en lo referente a la reducción de características es, de lejos, el GRASP. Esta victoria es debida a su comportamiento voraz en la generación de soluciones iniciales.

Con respecto a la tasa de clasificación, los resultados son bastante variables con respecto a las bases de datos, y tomar una decisión sobre la bondad de uno u otro sería aventurarse demasiado: esta decisión debería ser tomada teniendo en cuenta la base de datos particular con la que se quiera trabajar. Aunque la tasa de clasificación dentro de la muestra de entrenamiento es claramente mejor en GRASP, parece que estamos cayendo en la trampa del sobreajuste: al mirar la tasa en la muestra de test, GRASP pierde en comparación con los otros dos algoritmos —excepto en Arrythmia, donde es el ganador absoluto en todos los aspectos—.

Por tanto, si tuviera que escoger uno y solo uno de los algoritmos considerados, elegiría GRASP: su tasa de reducción es la mejor de los tres y las tasas de clasificación son bastante aceptables. Esta decisión vendría supeditada en casos reales, sin embargo, a un estudio pormenorizado del problema particular con el que estemos trabajando.